Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Implementação e Análise de Desempenho de Algoritmos de Difusão Atômica Tolerantes a Falhas Bizantinas

Vitor Mateus Costa do Rego

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Eduardo Adilio Pelinson Alchieri

Brasília
2019

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Edison Ishikawa

Banca examinadora composta por:

Prof. Dr. Eduardo Adilio Pelinson Alchieri (Orientador) — CIC/UnB
Prof. Dr. Marcelo Grandi Mandelli — CIC/UnB
Prof. Dr. Marcos Fagundes Caetano — CIC/UnB

CIP — Catalogação Internacional na Publicação

Rego, Vitor Mateus Costa do.

Implementação e Análise de Desempenho de Algoritmos de Difusão Atômica Tolerantes a Falhas Bizantinas / Vitor Mateus Costa do Rego. Brasília : UnB, 2019.
44 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2019.

1. difusão atômica, 2. paxos, 3. Collision-Fast, 4. USIG,
5. CFABCAST, 6. BCFABCAST, 7. USIG-BCFABCAST

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Implementação e Análise de Desempenho de Algoritmos de Difusão Atômica Tolerantes a Falhas Bizantinas

Vitor Mateus Costa do Rego

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Eduardo Adilio Pelinson Alchieri (Orientador)
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
CIC/UnB

Prof. Dr. Marcos Fagundes Caetano
CIC/UnB

Prof. Dr. Edison Ishikawa
Coordenador do Bacharelado em Ciência da Computação

Brasília, 13 de março de 2019

Dedicatória

Dedico esse trabalho inicialmente a Deus, pelo dom da vida e por permitir que eu chegasse até aqui. A minha família que sempre esteve presente e me ajudou sempre que necessário. Aos amigos por toda a ajuda e aos professores da UnB por todo o incentivo dado.

Agradecimentos

Gostaria de agradecer principalmente a minha família por estar sempre presente e apoiando minhas decisões, a Marcella por sempre acreditar em mim e me ajudar a me tornar uma pessoa melhor, aos meus professores por terem me ajudado a chegar até aqui, a todos os meus colegas de curso que me ajudaram muito ao longo desses 5 longos anos de UnB, e a todos que direta ou indiretamente me ajudaram até aqui.

A todos vocês, o meu mais sincero obrigado.

Resumo

O uso de protocolos baseado em Consenso é prejudicado pela presença de colisões (propostas concorrentes) que geram a necessidade de refazer essas propostas, isso leva a uma ineficiência do protocolo. O CFABCAST é um protocolo rápido, isso é, consegue decidir valores mesmo na presença de propostas concorrentes, porém ele não é tolerante a falhas bizantinas, por isso, [1] proporam o USIG-BCFABCAST, uma variação do CFABCAST que o consegue ser, porém esse protocolo não foi testado experimentalmente. Nesse trabalho fazemos uma análise experimental da eficiência desse protocolo e apresentamos os resultados obtidos.

Palavras-chave: difusão atômica, paxos, Collision-Fast, USIG, CFABCAST, BCFABCAST, USIG-BCFABCAST

Abstract

The use of protocols based on the Consensus is harmed because of the presence of collisions (concurrent proposals), this proposals then need to be re-proposed, causing a inefficacy of the protocol. The CFACBCAST is a collision fast protocol, which means it can be fast even in the presence of collision, but it is not Byzantine fault-tolerant. [1] proposed the USIG-BFABCAST, a protocol which is a modified version of the CFABCAST to handle Byzantine fault-tolerant, but this protocol has not be tested experimentally. On this paper we make a experimental analysis of the efficiency e present the results.

Keywords: atomic broadcast, paxos, Collision-Fast, USIG, CFABCAST, BCFABCAST, USIG-BCFABCAST

Sumário

1 Introdução **12**
1.1 Motivação . . . 12
1.2 Objetivos . . . 13
1.3 Organização . . . 14

2 Fundamentação Teórica **15**
2.1 Sistemas Distribuídos . . . 15
2.1.1 Modelos de Sistemas distribuídos . . . 15
2.1.2 Sincronismo entre Processos . . . 16
2.2 Agentes . . . 17
2.3 Consenso . . . 17
2.4 Difusão Atômica . . . 17
2.5 M-Consensus . . . 18
2.6 Paxos e variações . . . 19
2.7 CFABCAST - Collision-fast Atomic Broadcast . . . 23
2.8 Assinaturas digitais . . . 24
2.9 BCFABCAST - Bizantine Collision-fast Atomic Broadcast . . . 24
2.10 Componentes Seguros . . . 27
2.11 USIG-BCFABCAST - USIG-Bizantine Collision-fast Atomic Broadcast . . . 29

3 Implementação **32**
3.1 Linguagem utilizada . . . 32
3.2 Maven . . . 32
3.3 Netty . . . 32
3.4 Dynamic Quoruns . . . 33
3.4.1 QuorumMessage . . . 33
3.4.2 QuorumSender . . . 33
3.4.3 QuorumReplica . . . 33
3.4.4 Arquivos necessários . . . 33

3.5 Arquitetura 34
3.6 CFABCAST 34
3.7 BCFABCAST 35
3.8 USIG-BCFABCAST 36

4 Experimentos **38**
4.1 Cenários 38
4.2 Resultados 40

5 Conclusões e Trabalhos Futuros **42**
5.1 Visão geral 42
5.2 Revisão dos objetivos e contribuições 42
5.3 Perspectivas futuras 42

Referências **44**

Lista de Figuras

2.1 Modelo visual dos passos executados por cada agente [2] 23
2.2 Modelo de funcionamento da assinatura digital 25
2.3 Comparação entre a quantidade de passos do CFABACAST(esquerda) e do BCFABCAST(direita) [1] 27

3.1 Modelo da arquitetura da implementação do CFABCAST 35
3.2 Modelo da arquitetura da implementação do BCFABCAST 36
3.3 Modelo da arquitetura da implementação do USIG-BCFABCAST 37

4.1 Topologia utilizada 39
4.2 Latência obtida em milissegundos 40
4.3 Throughput obtido por segundo 41

Lista de Tabelas

2.1 Comparação entre algoritmos 30

Capítulo 1

Introdução

O uso de sistemas distribuídos está presente no cotidiano, grandes sistemas fazem uso dessa tecnologia, como por exemplo: Twitter [3], Facebook [4], Blockchaining [5], etc. E por esses sistemas serem descentralizados, isto é, ter cópias em vários locais, eles precisam garantir que todas as cópias possuem a mesma informação, para isso, eles utilizam protocolos de decisão, como o *Consensus* e a Distribuição Atômica.

1.1 Motivação

A Replicação de Máquina de Estados (SMR), do inglês *state machine replication*, é uma técnica muito utilizada para se obter redundância e aumento da disponibilidade de serviços, permitindo que esses serviços sejam tolerantes a falhas [6, 7]. Ela pode ser implementada a partir de primitivas de Difusão Atômica, ou ABCast do inglês*Atomic Broadcast* que permitem a entrega confiável e ordenada de mensagens a todos os destinatários não falhos.

O consenso distribuído, do inglês *Distributed Consensus*, é um algoritmo para que um conjunto de processos decida em um único valor. Este problema é equivalente a uma redução a Difusão Atômica [8]. Essa redução ocorre quando cada instância do Consenso decide um valor, e utilizando-se infinitas instâncias do Consenso e definindo uma ordenação entre elas, é estabelecida a ordem das mensagens entregues.

Quando são feitas propostas concorrentes, isto é, feitas ao mesmo tempo, pode ocorrer colisões, o que impede que uma das propostas seja decidida, e por causa disso, a proposta que foi perdida deve ser refeita. Os algoritmos que evitam essa reproposição são chamados de rápidos a despeito de colisões (*collision-fast*) e apresentam uma latência ótima de dois passos de comunicação, ou seja, não precisam de mais de dois passos comunicativos para chegar a uma decisão.

Na literatura não existem muitos algoritmos que sejam rápidos a despeito de colisões, isto é, possuem latência ótima de dois passos comunicativos mesmo na presença de colisões. Alguns dos algoritmos propostos na literatura são apresentados abaixo:

- [9] Mencius: o algoritmo proposto apesar de possuir apenas dois passos comunicativos em condições normais, não o é durante a falha de algum processo e o reinício do algoritmo.
- [10] Clock-RSM: o algoritmo faz uso de relógios sincronizados para conseguir a latência ótima, o que faz com que seja necessário uma resincronização dos relógios quando ocorre alguma falha, o que faz com que todo o protocolo tenha que esperar essa resincronização para continuar executando.
- [2] CFABCAST: o algoritmo é diferente dos anteriores por utilizar uma variação do Consenso chamada *M-Consensus* que consegue resolver o problema de difusão atômica sem impor penalidades como os algoritmos anteriores, ou seja, ele consegue manter a latência ótima de dois passos comunicativos mesmo quando ocorre colisões.

Todos os algoritmos, apesar de suas diferenças, procuram somente resolver o problema de falhas do tipo *crash*, não há entre eles nenhum que resolve o problema de falhas Bizantinas [11].

Por isso, Saramago, et al (2017) [1] proporam o BCFABCAST, um algoritmo capaz de continuar seu funcionamento mesmo quando ocorrem falhas bizantinas, ele utiliza como base o CFABCAST, porém não é rápido a despeito de falhas, por isso, eles também proporam o USIG-BCFABCAST que o consegue ser através do uso de um componente seguro no sistema.

Porém os algoritmos propostos por Saramago, et al (2017) [1] não foram verificados experimentalmente, o que pode atrapalhar ou mesmo impedir o seu uso em aplicações reais.

Essa monografia é continuação do trabalho realizado por Saramago, et al (2017) [1], fazendo a implementação desses algoritmos.

1.2 Objetivos

Implementar de maneira *opensource* os algoritmos BCFABCAST e USIG-BCFABCAST propostos em [1] dando continuidade a esse trabalho.

Além de implementar os algoritmos, compará-los experimentalmente com o o algoritmo CFABCAST.

1.3 Organização

Esta monografia está organizada da seguinte forma. No capítulo 2 será apresentada a fundamentação teórica em que esse trabalho se baseia, no Capítulo 3 será apresentado como foi feita a implementação do protocolo escrito em TLA+ para código executável, assim como discutidas as decisões tomadas durante o desenvolvimento. No Capítulo 4, será mostrado como os experimentos foram feitos, assim como os resultados obtidos. E por fim, no Capítulo 5, é apresentada a conclusão e possíveis trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Esse capitulo abordará toda a fundamentação teórica utilizada para o desenvolvimento dessa pesquisa, desde o que significa sistemas distribuídos, até como algumas das bibliotecas utilizadas funcionam.

2.1 Sistemas Distribuídos

Segundo [11] "Sistemas Distribuídos são uma coleção de computadores independentes que se mostram para o usuário como um único e coerente sistema". Essa definição nos permite entender que sistemas distribuídos dependem da existência de mais de um computador, justamente pelo termo distribuído, e que eles trabalham juntos para um propósito em comum.

2.1.1 Modelos de Sistemas distribuídos

O sistema distribuído, dependendo do foco que deseja obter, será criado usando modelos diferentes, estamos interessados em sistemas distribuídos que tenham foco em tolerância a falhas.

Tolerância a falhas

Um sistema tolerante a falhas é um sistema que quando um componente ou recurso que ele utiliza funcionava e por algum motivo parou de funcionar, o sistema deve estar preparado para lidar com esse tipo de situação [11]. Em outras palavras, pode-se dizer que o sistema deve conseguir se recuperar e continuar executando caso alguma situação anômala ocorra.

Esse trabalho se focará na tolerância a falhas de dois tipos: falhas do tipo crash, e falhas do tipo bizantina.

Falhas do tipo Crash

Essas falhas ocorrem quando o servidor ou sistema funcionava corretamente e por algum motivo ele para de funcionar, não fornecendo mais nenhuma resposta, ou seja, quando ele para de responder e não temos mais nenhuma informação dele.

Um bom exemplo, é quando você está acessando um site qualquer e o servidor desse site fica indisponível, fazendo com que o site não possa ser acessado enquanto o servidor estiver fora do ar.

Falhas Bizantinas

O termo bizantino vem do antigo Império Bizantino, conhecido por ser um lugar com muita conspiração, intriga e desconfiança [11]. Esse tipo de falha é quando um processo ou servidor fornece respostas que não são esperadas, isto é, são propositalmente incorretas ou que levem a resultados não esperados, sendo essa a principal razão do perigo desse tipo de falha, pois o sistema permanece funcional, mas as ações dele leva a resultados não esperados ou incorretos.

Um exemplo é quando um programa foi infectado por um vírus e continua executando normalmente para o usuário, mas executa instruções que o usuário não solicitou.

2.1.2 Sincronismo entre Processos

Muitos processos precisam se comunicar entre si, e existem duas maneiras de se conseguir isso:

- Memória Compartilhada
- Troca de Mensagens

Como os Sistemas Distribuídos tem foco em processos existentes em maquinas diferentes, trabalharemos apenas com **troca de mensagens**. Ela utiliza duas primitivas *send* e *receive*, em que a primeira envia uma mensagem para um destino e a segunda recebe uma mensagem de uma fonte qualquer (Tanenbaum, 2003) [12]. Essa forma de comunicação permite sincronizar os processos, ou seja, garantir que um processo só execute uma ação após receber uma mensagem para executá-la, ou o contrário, onde ele só pode executar uma ação após enviar uma mensagem.

O sincronismo está relacionado com a comunicação entre os processos, que basicamente são de dois tipos: a comunicação síncrona e a comunicação assíncrona.

Comunicação síncrona

Esse caso é quando um processo envia uma mensagem para outro e espera até que o outro processo informe que recebeu a mensagem, ou seja, ele só continua sua execução quando garante que o processo destinatário recebeu sua mensagem.

Comunicação assíncrona

Esse caso é quando um processo envia uma mensagem para outro processo e não espera para verificar se ela foi recebida, ou seja, ele envia a mensagem e continua sua execução, sem esperar para saber se ela foi recebida.

2.2 Agentes

Nesse trabalho utilizamos uma abstração chamada de agentes, agentes são entidades que executam uma tarefa computacional (processos, threads, atores, etc), eles se comunicam utilizando troca de mensagens com comunicação assíncrona, onde não existe tempo limite para as ações realizadas pelos agentes.

A abstração de agente será utilizada nas definições a seguir, assim como ao longo de todo o trabalho desenvolvido.

2.3 Consenso

O consenso é um problema clássico de sistemas distribuídos. O problema consiste em um conjunto de agentes, onde os agentes propõem valores, e ao fim decidem em um dos valores propostos. A decisão deve ser aprendida por todos os agentes corretos relacionados. Aprender significa receber e utilizar a informação recebida. Um agente é considerado correto se ele não é falho, e ele é falho se sua execução foge da especificação do protocolo, podendo inclusive executar ações maliciosas (bizantinas).

2.4 Difusão Atômica

Dados dois conjuntos de agentes, *proposers* e *learners*, o problema da Difusão atômica consiste em garantir que todas as mensagens enviadas pelos *proposers* cheguem aos *learners*, na mesma ordem. Como dito em [13] podemos entender como acordo, uma sequência crescente de mensagens difundidas por *proposers*, em que os *learners* aprendem incrementando prefixos nessa sequência.

A Difusão Atômica é definida pelas seguintes propriedades, onde *recebidos[l]* é uma sequência de mensagens entregues por um *learner* l, inicialmente vazio.

Não-trivialidade para qualquer *learner* l, recebidos[l] contém apenas mensagens enviadas por *proposers* e nenhuma duplicata.

Estabilidade para qualquer *learner* l, se recebidos[l] = s em algum momento, então s $\subseteq$ recebidos[l] para qualquer instante posterior.

Consistência para dois *learners* l1 e l2, ou recebidos[l1] está contido em recebidos[l2] ou o contrário.

Em sistemas reais, os clientes enviam uma mensagem e aprendem o resultado dessa execução, podemos associar essas ações com os *proposers* e os *learners*. Porém não podemos garantir que os clientes não são falhos, portanto definimos o progresso do protocolo a partir de um outro quorum de agentes, os *acceptors*. Definimos o progresso do protocolo da seguinte forma:

Progresso Para qualquer *proposer* p, *learner* l e um quorum de *acceptors* não falhos e p envia uma mensagem m, então eventualmente recebidos[l] terá m.

A Difusão Atômica e Consenso são problemas equivalentes, sendo possível resolver o consenso a partir de uma redução à difusão atômica e vice-e-versa [8].

2.5 M-Consensus

O M-Consensus é uma variante do consenso, nele, múltiplos valores podem ser decididos em uma única instância. [2] provaram como fazer a redução da Difusão Atômica para o M-Consensus. Explorando o fato de poder decidir vários valores em uma única instância, eles conseguiram obter um algoritmo rápido a despeito de falhas do tipo crash.

Neste protocolo, os agentes devem decidir em um mapeamento de *proposers* para valores propostos ou um valor especial *Nil*. Esse mapeamento é registrado em um mapa de valores, chamado *v-mapping*, que é um conjunto, potencialmente vazio, de funções que mapeiam proposers a valores propostos ou *Nil*.

Os mapas possuem uma relação de precedência, podendo a partir disso estabelecer uma ordem total entre eles (utilizando um identificador único para cada *proposer*). Um *v-map* com apenas um elemento em seu domínio é dito simples ou *single-mapping*. Ele é dito trivial se todos os *proposers* em seu domínio mapeiam para o valor *Nil*. Dois *v-mapping* diferentes são ditos compatíveis se os elementos na intersecção de seus domínios mapeiam para os mesmos valores, portanto, esses mapas podem ser estendidos até que se tornem

iguais, pois não há discordância entre eles. Logo, em um algoritmo de M-Consenso, os *proposers* propõem valores e os *learners* aprendem um *v-map*, possivelmente diferentes, mas sempre compatíveis, e que podem ser estendidos até que sejam iguais.

2.6 Paxos e variações

O Paxos é um protocolo de consenso proposto por [14] e conhecido por seu uso na indústria, como por exemplo o *Chubby Lock Service* [15]. Nele, as propostas (comandos) são enviados a partir do cliente para um *proposer* qualquer, porém somente o coordenador, um *proposer* com permissões especiais, pode propor um valor para ser decidido pelos *acceptors*. E não existindo mais de um agente que se julgue coordenador, junto à um quorum correto e alcançável de *acceptors* então o protocolo efetivamente chegará em um resultado.

Cada instância de execução do Paxos decide em um valor, podendo cada instância ter infinitas rodadas, que são criadas caso ocorra alguma falha em se decidir o valor na rodada anterior. Cada instância possui apenas um único *proposer* que atua como coordenador, e apenas ele pode propor um valor. Caso ele falhe, outro *proposer* pode ser eleito coordenador, a partir de algum protocolo a parte, e uma nova rodada é iniciada para essa instância.

O algoritmo proposto por [14] pode ser visto no Algoritmo 1. Os algoritmos estão apresentados na linguagem TLA+, que é uma linguagem para modelagem de sistemas distribuídos [16].

O Paxos é executado em rodadas, cada rodada com duas fases, sendo a primeira fase de configuração do protocolo, e a segunda fase é a que efetivamente leva a uma decisão. Em um caso o qual múltiplas instâncias são necessárias, a primeira fase pode ser executada em paralelo para todas as instâncias, o que diminui o custo, e a decisão ocorre somente com a fase 2.

A primeira etapa da primeira fase é chamada *Phase1a*, onde o coordenador envia uma mensagem para os *acceptors*, essa mensagem contém a rodada atual r, e é chamada de *msg1a*, e o *acceptor*, ao recebê-la, pode executar uma das seguintes ações:

- Caso não tenha recebido nenhuma *msg1a* anteriormente, ele irá atualizar a rodada atual para r, e enviar uma mensagem de confirmação *msg1b* para o coordenador, para garantir que não aceitará valores que possuam uma rodada menor que r.
- Caso já tenha recebido uma mensagem *msg1a* em uma rodada $s < r$ e não recebeu nenhuma mensagem *msg2A* com uma proposta enviada durante a fase de decisão pelo coordenador, então ele atualiza a própria rodada para a maior rodada recebida, e envia a mensagem de confirmação *msg1b* para o coordenador.

Algorithm 1 Collision Fast Paxos [Schmidt et al 2014] [2]

Pr, *A*, *L*: proposers, acceptors and learners sets;
$CF(i)$: round i's collision-fast proposers set;
$C(i)$: round i's coordinator.
prnd[*p*], *crnd*[*c*], *rnd*[*a*]: current rounds of proposer p, coordinator c, and acceptor a, respectively, initially 0.
pval[*p*]: value p has fast-proposed at *prnd*[*p*] or *none* if p has not fast-proposed at *prnd*[*p*], initially *none*.
cval[*c*]: initial v-mapping for *crnd*[*c*], if c has queried an acceptor quorum or *none* otherwise; initially $\perp$ for coordinator of round 0 and *none* for others.
vrnd[*a*]: round at which a has accepted its latest value.
vval[*a*]: v-mapping a has accepted at *vrnd*[*a*] or *none* if no value has been accepted at *vrnd*[*a*]; initially *none*.
learned[*l*]: v-mapping currently learned by learner l; initially $\perp$.

Propose$(p, V) \triangleq$
  pre-condition:
    $p \in Pr$
  action:
    send $\langle$"`propose`", $V\rangle$ to $cf \in CF(prnd[p])$
Phase1a$(c, r) \triangleq$
  pre-conditions:
    $c = C(r)$
    $crnd[c] < r$
  actions:
    $crnd[c] \leftarrow r$
    $cval[c] \leftarrow none$
    send $\langle$"`1a`", $r\rangle$ to A
Phase1b$(a, r) \triangleq$
  pre-conditions:
    $a \in A$
    $rnd[a] < r$
    received $\langle$"`1a`", $r\rangle$ from $C(r)$
  actions:
    $rnd[a] \leftarrow r$
    send $\langle$"`1b`", $a, r, vrnd[a], vval[a]\rangle$ to $C(r)$
Phase2Start$(c, r) \triangleq$
  pre-conditions:
    $c = C(r)$
    $crnd[c] = r$
    $cval[c] = none$
    $\exists Q$: Q is a quorum : $\forall a \in Q$ received $\langle$"`1b`", $a, r, vrnd, vval\rangle$
  actions:
    LET $msgs = [m = \langle$"`1b`", $a, r, vrnd, vval\rangle$: received m from a $\in$ Q]
    LET $k = Max([vrnd : \langle$"`1b`", $a, r, vrnd, vval\rangle \in msgs$])
    LET $S = [vval : \langle$"`1b`", $a, r, k, vval\rangle$ $\in$ m, vval $\neq none]$
    IF $S = \emptyset$ THEN
      $cval[c] \leftarrow \perp$
      send $\langle$"`2S`", $r, cval[c]\rangle$ to P
    ELSE
      $cval[c] \leftarrow \sqcup S \bullet [\langle p, Nil\rangle : p \in Pr]$
      send $\langle$"`2S`", $r, cval[c]\rangle to P \cup A$
Phase2Prepare$(p, r) \triangleq$
  pre-conditions:
    $p \in Pr$
    $prnd[p] < r$
    $\langle$"`2S`", $r, v, proofs\rangle_{C(r)} \Leftarrow C(r)$
    goodRoundValue(r,v,proofs)
  actions:
    $prnd[p] \leftarrow r$
    $proof[p] \leftarrow proofs$
    IF $v = \perp$ THEN $pval[p] \leftarrow none$
    ELSE $pval[p] \leftarrow v(p)$
Phase2a$(p, r, V) \triangleq$
  pre-conditions:
    $p \in CF(r)$
    $prnd[p] = r$
    $pval[p] = none$
    either ($V \neq Nil$ **and** received $\langle$"`propose`", $p, V\rangle$)
    or ($V = Nil$ **and** received $\langle$"`2a`", $r, \langle q, W\rangle\rangle$, $q \in CF(r)$, $W \neq Nil$)
  actions:
    $pval[p] \leftarrow V$
    IF $V \neq Nil$ THEN
      send $\langle$"`2a`", $p, r, \langle p, V\rangle\rangle$ to $A \cup CF(r)$
    ELSE
      send $\langle$"`2a`", $p, r, \langle p, V\rangle\rangle$ to L
Phase2b$(a, r) \triangleq$
  LET *Cond1* = (received $\langle$"`2s`", $r, v\rangle$, $v \neq \perp$ and vrnd[a] < r) or vval[a] = none
  LET *Cond2* = received $\langle$"`2a`", $r, \langle p, V\rangle\rangle$, $V \neq Nil$
  pre-conditions:
    $a \in A$
    $rnd[a] \leq r$
    either *Cond1 or Cond2*
  actions:
    IF *Cond1*
      THEN $vval[a] \leftarrow v$
      ELSE
        IF *Cond2 and* ($vrnd[a] < r$ *or* $vval[a] = none$)
          THEN $vval[a] \leftarrow \perp \bullet \langle p, V\rangle \bullet [\langle p, Nil\rangle : p \in Pr \setminus CF(r)]$
          ELSE $vval[a] \leftarrow vval[a] \bullet \langle p, V\rangle$
    $rnd[a] \leftarrow vrnd[a] \leftarrow r$
    send $\langle$"`2b`", $r, vval[a]\rangle$ *to* L
Learn$(l) \triangleq$
  pre-conditions:
    $l \in learners$
    $\exists Q \subseteq A$:
      Q is a quorum
      $\forall a \in Q$, $\langle$"`2b`", $r, \_\rangle_a \Leftarrow a$
  actions:
    $Q2bVals = \{v : \langle$"`2b`", $r, v\rangle_a \Leftarrow a \in Q\}$
    $w = \sqcap Q2bVals$
    $learned[l] = learned[l] \sqcup w$

- Caso já tenha recebido uma mensagem *msg1a* em uma rodada s < r e recebeu alguma mensagem *msg2A* com uma proposta, ele envia essa proposta junto a rodada em que foi votado na mensagem *msg1b* para o coordenador.

A fase de decisão (*Phase2*) só ocorre após o coordenador ter recebido mensagens de um quorum de *acceptors* durante a *Phase1*. Com as mensagens *msg1B* recebidas, o coordenador pode determinar qual valor será enviado na mensagem *msg2A* que será proposto, com base nos seguintes critérios:

Caso ele tenha recebido uma ou mais mensagens *msg1b*, ele escolhe a proposta que tenha o maior valor de rodada.

Se nenhuma das mensagens *msg1b* contém um valor aceito, então ele pode escolher qualquer valor para propor.

Ao selecionar um valor, ele envia a mensagem *msg2A* com o valor e a rodada atual para um quorum de *acceptors*. Um *acceptor*, ao receber a mensagem *msg2a* aceita o valor proposto somente se ele respondeu anteriormente a uma mensagem *msg1a*, o *acceptor* então envia uma mensagem *msg2b* com o valor aceito para um quorum de *learners*.

É importante perceber que quando um *acceptor* aceita um valor, não significa que esse valor é aprendido pelos *learners*, eles só aprendem um valor quando um quorum de *acceptors*, isto é, uma maioria dos *acceptors* aceitarem o mesmo valor proposto.

A *Phase1* só é reexecutada em caso de suspeita de falha na rodada atual, o que exige que o coordenador crie uma nova rodada. A *Phase1* também pode ser executada paralelamente para múltiplas intâncias. Com isso, durante uma execução normal do protocolo, os clientes enviam requisições aos *proposers* que as encaminham para o coordenador que executa a *Phase2* do protocolo, obtendo um valor decidido ao final de cada instância, e respondendo o cliente. Portanto é a *Phase2* que apresenta custos reais de passos comunicativos no protocolo, precisando então de 3 passos de comunicação.

Em [17] é proposto o Fast-Paxos, esse algoritmo consegue obter uma latência de dois passos de comunicação, a latência é o tempo de resposta entre e pedido do cliente e sua confirmação, ele consegue isso fazendo com que os *proposers* façam suas propostas diretamente aos *acceptors*, sem ter que passar pelo coordenador. O Fast-Paxos funciona da mesma maneira que o Paxos, isto é, em fases, sua diferença ocorre na *Phase2*, onde está a configuração que permite que o *proposer* envie as propostas diretamente para os *acceptors*, isso remove um passo comunicativo, que é o de envio da proposta para o coordenador.

Porém, [18] provou que o Paxos é ótimo, então para poder reduzir a latência, precisamos sacrificar a resiliência, com isso, precisamos de um total de 3f + 1 acceptors para tolerar f acceptors falhos e garantir consistência, fazendo com que os *learners* precisem

do aceite de pelos menos 2f+1 *acceptors* para aprender um valor. Já o Paxos precisa que apenas f+1 *acceptors* aceitem um valor.

O Paxos precisa de 3 (três) passos comunicativos, já o Fast-Paxos precisa de 2 (dois) passos, apesar de ser mais eficiente, nenhum deles é rápido a despeito de falhas. Por isso, [2] proporam o Collision-Fast Paxos que consegue ser rápido mesmo na presença de falhas do tipo *crash*. Ele resolve o problema das colisões ao solucionar uma instância do M-Consensus e como o próprio nome diz, ele é Collision-Fast, ou seja, consegue ser executado com uma latência ótima de dois passos comunicativos.

Nele as rodadas são totalmente ordenadas por uma relação de precendência e contém informação sobre quem é o coordenador e qual o conjunto de *Collision-fast Proposers dela*.Os *Collision-fast proposers* são um subconjunto do conjunto de *proposers* de uma rodada, e eles são os únicos *proposers* que podem ter suas propostas decididas em dois passos comunicativos (conhecido como proposta rápida). Eles podem fazer uma proposta rápida quando:

- possui um valor V $\neq$ Nil, isto é, quando ele recebeu um valor do cliente para propor, ou
- percebe que um outro *CF-proposer* propôs de forma rápida um valor diferente de Nil, e ele, o *CF-proposer p* então propõe de forma rápida *<p, Nil>*.

Caso essa proposta possua um mapa com valor V diferente de Nil, então o *CF-proposer* envia V para o quorum de *acceptors* e demais *CF-proposers* da rodada, caso contrário, ele envia o valor Nil diretamente para os *learners*.

Um *acceptor* pode aceitar múltiplos *v-mappings* (mapa de valores) desde que os novos mapas estendam os previamente aceitos. Os *v-mappings* aceitos pelos *acceptors* são construídos a partir de um *s-mapping não-trivial* que tenha sido proposto de forma rápida, logo, sempre possuem um mapa que mapeia pelo menos um *proposer* para um valor diferente de *Nil*, o que garante a não trivialidade da decisão.

Dessa forma, um *v-mapping* é aceito em uma rodada r se ele é prefixo de todo *v-mapping* aceito por um quorum de *acceptors* em r. Portanto, tal *v-mapping* é compatível e um *learner* pode estender *learned[l]*, aprendendo tal *v-mapping*.

Se um *v-mapping* foi aprendido em uma rodada, então sempre é possível estendê-lo para todas as rodadas seguintes. Isso é garantido pelas ações tomadas pelo coordenador no início de uma nova rodada.

O coordenador, ao criar uma nova rodada, consulta um quorum de *acceptors* para verificar se algum *v-mapping* foi ou pode ser aceito em alguma rodada anterior. Se existir algum, ele pode completá-lo com valores *Nil* para torná-lo completo e enviar para os *acceptors* para que possa ser aceito e depois aprendido. Caso nenhum *v-mapping* tenha

sido aceito nessas rodadas anteriores, ele informa os *CF-proposers* da rodada atual que eles podem propor de forma rápida nessa rodada.

A Figua 2.1 mostra visualmente qual a ordem dos passos do protocolo, e qual agente é responsável por cada passo.

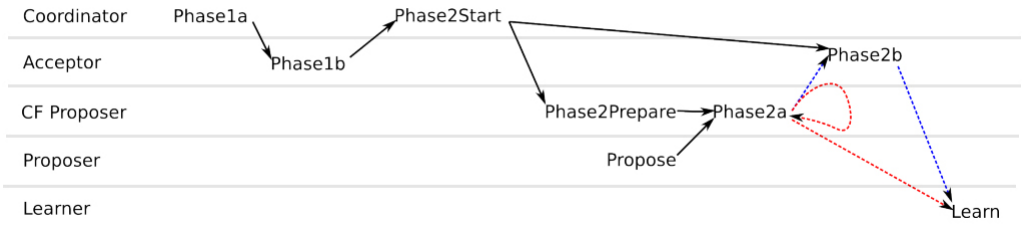

Figura 2.1: Modelo visual dos passos executados por cada agente [2]

2.7 CFABCAST - Collision-fast Atomic Broadcast

O *Collision-Fast Paxos* pode ser utilizado para implementação de um protocolo de difusão atômica utilizando sucessivas instâncias do *M-Consensus*, assim como é feito na redução da difusão atômica para o consenso. Como várias mensagens podem fazer parte de cada decisão do *Collision-Fast Paxos*, várias mensagens podem ser entregues ao fim de cada instância, em dois passos comunicativos. O CFABCAST é quem controla a resolução dessas infinitas instâncias do Collision-Fast Paxos que estão sendo realizadas.

Assim, ao invés de solucionar infinitas instâncias do *Consensus*, essa implementação resolve infinitas instâncias de sua variante, o *M-Consensus*. Nele, cada instância é unicamente identificada por um número natural i e a solução de cada instância forma uma sequência de comandos de *Broadcast*.

Em um sistema de máquinas de estados replicadas implementado com base no protocolo de Difusão Atômica, clientes fazem o *broadcast* de comando para réplicas que as enviam e, de forma determinista, executam a sequência totalmente ordenada de comandos decididos, permitindo que todas as réplicas cheguem à um mesmo estado. O desempenho do sistema depende do desempenho do protocolo de Difusão Atômica, assim, um protocolo que permita a entrega de comandos em menor número de passos comunicativos permite que o desempenho desses sistemas seja melhor. O Algoritmo 2 mostra a implementação do algoritmo em TLA+.

Algorithm 2 Collision-fast Atomic Broadcast [Schmidt et al 2014] [2]

I: the set of all Collision-fast Paxos instances used $CFP(i)!A$: the action or variable A of Collision-fast Paxos instance i

1: $Propose(p, V) \triangleq$
2: $\quad \forall i \in I, CFP(i)!Propose(p, V)$
3: $NewPhase1a(i, c, r) \triangleq$
4: **pre-conditions:**
5: $\quad c = C(r)$
6: $\quad crnd[c] < r$
7: $\quad c$ believes itself to be the leader
8: $\quad c$ heard of a round $r > j > crnd[c]$ for some instance or $CF(crnd[c]) \notin active[c]$
9: **actions:**
10: $\quad CFP(i)!Phase1a(c, r)$
11: $Phase1a(c, r) \triangleq$
12: $\quad \forall i \in I, CFP(i)!NewPhase1a(i, c, r)$
13: $Phase1b(a, r) \triangleq$
14: $\quad \forall i \in I, CFP(i)!Phase1b(a, r)$
15: $Phase2Start(c, r) \triangleq$
16: $\quad \forall i \in I, CFP(i)!Phase2Start(c, r)$
17: $Phase2Prepare(p, r) \triangleq$
18: $\quad \forall i \in I, CFP(i)!Phase2Prepare(p, r)$
19: $Phase2a(p, r, V) \triangleq$
20: **pre-condition:**
21: $\quad p$ has not yet proposed V
22: **action:**
23: $\quad$ LET $i = Min([j : CFP(j)!pval[p] = none])$
24: $\quad CFP(i)!Phase2a(p, r, V)$
25: $Phase2b(i, a, r) \triangleq$
26: $\quad CFP(i)!Phase2b(a, r)$
27: $Learn(i, l) \triangleq$
28: $\quad CFP(i)!Learn(l)$

2.8 Assinaturas digitais

A assinatura digital é um método de autenticação digital, ou seja, uma maneira de garantir a origem e integridade de uma informação enviada em meio digital. Geralmente utilizando criptografia de chaves assimétricas para que possa ocorrer uma comunicação confiável.

De maneira simplificada, o processo funciona da seguinte maneira: aplica-se uma função *hash* (ex: SHA-1, SHA-256, MD5, etc) a mensagem que se deseja enviar, o resultado obtido é então criptografado utilizando um sistema de chave pública. A mensagem então é assinada junto ao hash gerado, pela chave privada do remetente.

O destinatário, ao receber a mensagem, deve verificar sua autenticidade, para isso, ele aplica a mesma função hash na mensagem recebida, e o compara com o hash da mensagem, obtido a partir da decifração da mensagem utilizando a chave pública do remetente. Caso os hashes obtidos sejam iguais, então ele possui a confirmação de que a mensagem foi enviada a partir do remetente esperado. O processo é mostrado na Figura 2.2.

2.9 BCFABCAST - Bizantine Collision-fast Atomic Broadcast

O BCFABCAST resolve o M-Consenso na presença de falhas bizantinas [1]. Porém, apesar do nome e ser derivado do *CFABCAST* ele não é rápido a despeito de colisões. Para ser

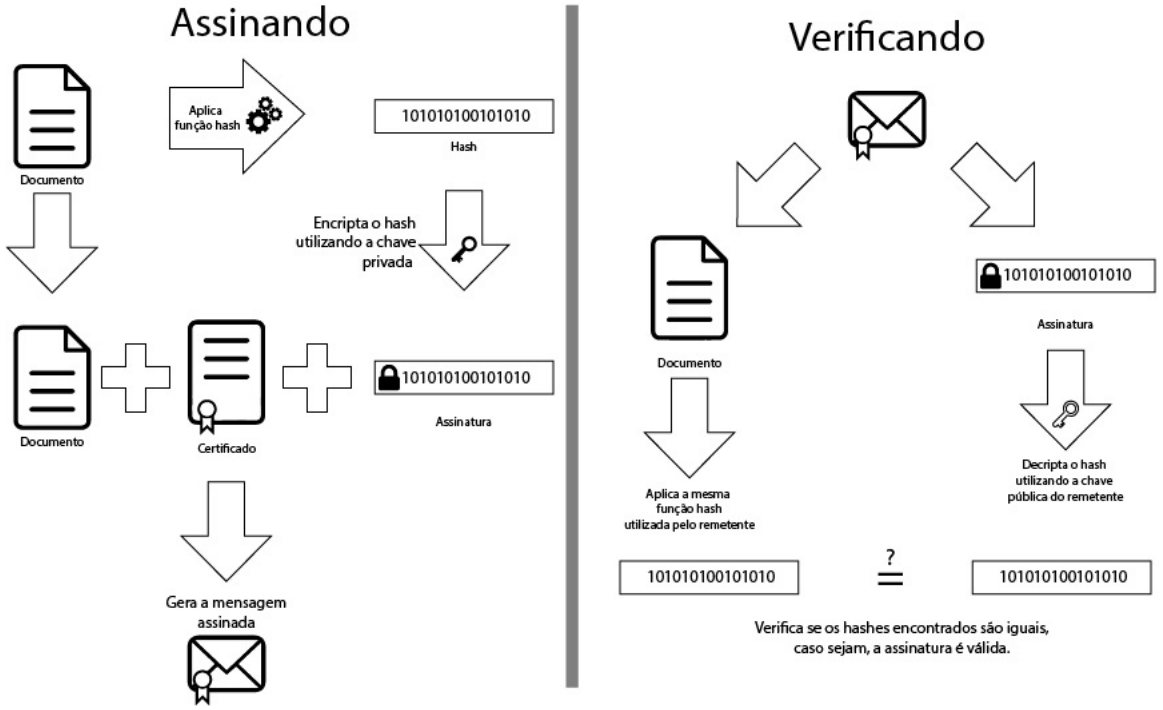

Figura 2.2: Modelo de funcionamento da assinatura digital

tolerante as falhas bizantinas, ele utiliza assinatura digital no envio de suas mensagens.

Assim como as outras variantes do Paxos, o algoritmo é executado em rodadas, dividido em duas fases. Na fase 1 o coordenador procura por valores aceitos em rodadas anteriores com os *acceptors* (ação Phase1a). Baseado na resposta fornecida pelos *acceptors* (ação Phase1b) o coordenador estabelece se algum *v-map* já foi decidido anteriormente (ação Phase2Start). O protocolo garante que se algum mapeamento tiver sido decidido, então pelo menos 2f+1 mensagens *1b* possuirão esse mapeamento. Seja S o conjunto de todos os *single-maps* aceitos por 2f+1 acceptors, se S não é vazio, então o coordenador informa os *CF-proposers* para que se abstenham de propor nessa rodada, e os *acceptors*, para que aceitem os mapeamentos em S e informem os *learners* (ação Phase2b). Com isso, o coordenador tenta terminar a instância e progredir no protocolo.

Caso contrário, o coordenador informa os *CF-proposers* da rodada sendo iniciada, para que proponham seus valores para os *acceptors* (ação *Phase2Prepare*). Autorizado pelo coordenador, um *CF-proposer* pode repassar o valor em mensagem recebida de um *proposer* qualquer, incluindo a si mesmo (ação *Phase2a*). Os *CF-proposers* então passam a mensagem *2S* recebida do coordenador para os *acceptors* para mostrarem que possuem permissão de propor.

Para garantir que o coordenador não possa executar ações bizantinas em relação ao consenso, ele deve provar que executou a fase 1 corretamente. Isso é feito enviando o conjunto de mensagens recebidas por um quorum de *acceptors* e usadas para computar

S nas mensagens *2S*. Em seguida os agentes receptores realizam o mesmo cálculo para comprovar que o valor proposto realmente é válido para a rodada em questão (função *goodRoundValue*).

As alterações feitas no algoritmo servem apenas para coibir ações de agentes bizantinos, fazendo com que o algoritmo herde a corretude do CFABCAST. Focando então em coibir as ações dos *acceptors* e dos *CF-proposers*, isso porquê os *learners* não influenciam na execução de outros agentes, pois não enviam mensagens, contudo podem aprender qualquer valor, sendo esse um problema incontornável. Já os *proposers* possuem apenas uma ação, que é propor valores, e para fazer isso devem seguir o protocolo, não influenciando o protocolo mesmo se forem maliciosos.

Assim, sendo f o número de falhas, em um total de 5f acceptors, 4f+1 devem ser corretos. Com isso, em um quorum de 4f+1, garantidamente 3f+1 *acceptors* serão corretos. Os *learners* esperam por 4f+1 mensagens de confirmação com o mesmo mapeamento antes de aprendê-lo, aprendendo, portanto, somente valores aceitos por um quorum de *acceptors*. Se tiver *acceptors* e *CF-proposers* bizantinos, na pior das hipoteses, eles podem entrar em conluio para aceitarem valores diferentes, repassados a *learners* diferentes. Assim, apesar de esse ataque impedir uma rodada de terminar, ele não pode levar a decisões conflitantes.

Se o coordenador for bizantino ele poderia ignorar as mensagens recebidas dos *acceptors* durante a fase 1 numa tentativa de violar a propriedade de acordo do protocolo. Por isso, o protocolo exige que o coordenador prove que executou a fase 1 corretamente. Isso é feito a partir das mensagens *msg1b* assinadas, que foram previamente recebidas, e o coordenador não pode forjar. Portanto, o coordenador não consegue manipular o que já foi aprendido anteriormente sem ser percebido, apesar de conseguir forçar reconfigurações corretas indefinidamente, o que compromete o progresso do protocolo. Esse problema pode ser resolvido utilizando rodízio do papel de coordenador, o que limita o ataque.

No CFABCAST, quando um *CF-proposer p* não possui um valor para propor e recebe uma proposta de outro *CF-proposer k*, *p* envia uma mensagem diretamente para os *learners* avisando que não irá propor nessa rodada. O *learner*, ao receber essa mensagem de *p*, aprende que *p* não pode ser mapeado a outro valor, não precisando do aceite dos *acceptors*. Se a mesma abordagem fosse possível no BCFABCAST, então um *CF-proposer* poderia dizer que está se abstendo de propor e ao mesmo tempo enviando uma proposta aos *acceptors*.

Para impedir isso, no BCFABCAST, um *CF-proposer* não pode se comunicar diretamente com os *learners*. E para impedir que um *CF-proposer* fique propondo *Nil* continuamente, é exigido que ele só proponha caso tenha recebido uma mensagem *propose* ou *msg2a*. Com isso, caso o *CF-proposer* não tenha nada a propor, serão necessários até três passos de comunicação para terminar o protocolo: um passo para receber a mensagem

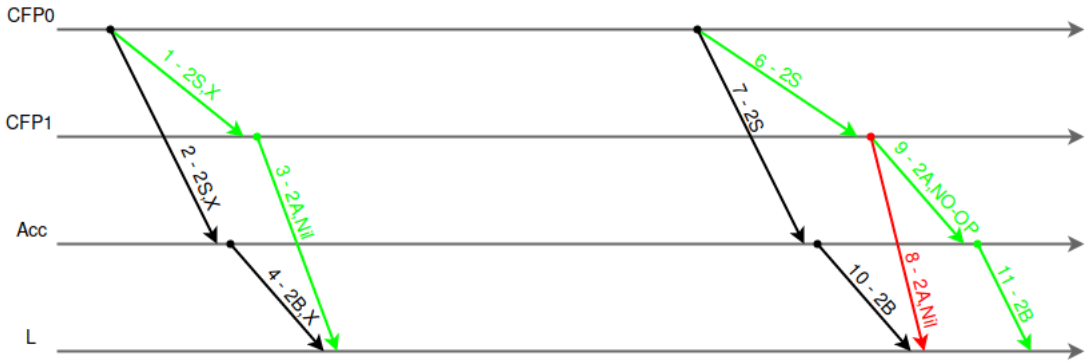

Figura 2.3: Comparação entre a quantidade de passos do CFABACAST(esquerda) e do BCFABCAST(direita) [1]

msg2a e mais dois passos para enviar a abstenção aos *learners* pelos *acceptors*. Na Figura 2.3 podemos ver claramente como ocorrem os passos comunicativos. O Algoritmo 3 mostra a implementação do algoritmo em TLA+.

2.10 Componentes Seguros

Componente seguro, no contexto de falhas bizantinas, é um componente que garante que a comunicação entre agentes não pode ser alterada por agentes bizantinos, ou seja, garante que um agente bizantino não terá poder para alterar a decisão do quorum.

Os componentes seguros utilizados no desenvolvimento de protocolos tolerantes a falhas bizantinas possuem diferenças tanto em aspectos de implementação quanto em desempenho. Quanto a implementação, eles podem ser:

- Implementados de forma distribuída, por exemplo o *Trusted Timely Computing Base* (TTCB) [19]
- Locais, baseando em memória, por exemplo o *Attested Append-Only Memory* (A2M) [20]
- Locais, baseados em um contador, por exemplo o *Unique Sequential Identifier Generator* (USIG) [21].

Um atacante não consegue acessar estes componentes, mesmo conseguindo acessar um servidor [1]. Com isso, é possível projetar protocolos que limitam as ações que um agente bizantino pode executar sem ser descoberto. Nesse trabalho, usaremos o componente seguro USIG, esse componente é um serviço local a cada agente, ou seja cada um vai ter a sua própria implementação do USIG.

O componente USIG adiciona as mensagens enviadas um contador, e assina as mensagens com esse identificador. Cada identificador é único, monotônico e sequêncial para

Algorithm 3 Byzantine M-Consensus [Saramago et al 2017] [1]

Pr, A, L: proposers, acceptors and learners sets;
$CF(i)$: round i's collision-fast proposers set;
$C(i)$: round i's coordinator.
$prnd[p]$, $crnd[c]$, $rnd[a]$: current rounds of proposer p, coordinator c, and acceptor a, respectively, initially 0.
$pval[p]$: value p has fast-proposed at $prnd[p]$ or $none$ if p has not fast-proposed at $prnd[p]$, initially $none$.
$cval[c]$: initial v-mapping for $crnd[c]$, if c has queried an acceptor quorum or $none$ otherwise; initially $\perp$ for coordinator of round 0 and $none$ for others.
$vrnd[a]$: round at which a has accepted its latest value.
$vval[a]$: v-mapping a has accepted at $vrnd[a]$ or $none$ if no value has been accepted at $vrnd[a]$; initially $none$.
$learned[l]$: v-mapping currently learned by learner l; initially $\perp$.
$m \Leftarrow s$: received message m from source s
$m \Rightarrow d$: send message m to destination d

$Propose(p, V) \triangleq$
  pre-condition:
    $p \in Pr$
  action:
    $\langle$"propose", $V\rangle_p \Rightarrow cf \in CF(prnd[p])$

$Phase1a(c, r) \triangleq$
  pre-conditions:
    $c = C(r)$
    $crnd[c] < r$
  actions:
    $crnd[c] \leftarrow r$
    $cval[c] \leftarrow none$
    $\langle$"1a", $r\rangle_c \Rightarrow A$

$Phase1b(a, r) \triangleq$
  pre-conditions:
    $a \in A$
    $rnd[a] < r$
    $\langle$"1a", $r\rangle_c \Leftarrow C(r)$
  actions:
    $rnd[a] \leftarrow r$
    $\langle$"1b", $r, vrnd[a], vval[a]\rangle_a \Rightarrow C(r)$

$Phase2Start(c, r) \triangleq$
  pre-conditions:
    $c = C(r)$
    $crnd[c] = r$
    $cval[c] = none$
    $\exists Q \subseteq A$:
      Q is a quorum
      $\forall a \in Q$, $\langle$"1b", $r, vrnd, vval\rangle_a \Leftarrow a$
  actions:
    LET $1bs = [m = \langle$"1b", $\_, \_, \_\rangle_a : m \Leftarrow a \in Q\ ]$
    LET $k = Max([vrnd : \langle$"1b", $\_, vrnd, \_\rangle_a \in 1bs\ ])$
    LET $A_{\langle p,v\rangle} = [a : \langle$"1b", $r, k, vval\rangle_a \in 1bs \wedge vval[p] = v]$
    LET $S = \left[\langle p, v\rangle : \left|A_{\langle p,v\rangle}\right| \geq 2f + 1\right]$
    IF $S = \emptyset$ THEN
      $cval[c] \leftarrow \perp$
    $\langle$"2S", $r, cval[c], msgs\rangle_c \Rightarrow CF(r)$
    ELSE
    $cval[c] \leftarrow \sqcup S \bullet [\langle p, Nil\rangle : p \in CF(r)]$
    $\langle$"2S", $r, cval[c], msgs\rangle_c \Rightarrow CF(r) \cup A$

$Phase2Prepare(p, r) \triangleq$
  pre-conditions:
    $p \in CF(r)$
    $prnd[p] < r$
    $\langle$"2S", $r, v, proofs\rangle_{C(r)} \Leftarrow C(r)$
    goodRoundValue(r,v,proofs)
  actions:
    $prnd[p] \leftarrow r$
    $proof[p] \leftarrow proofs$
    IF $v = \perp$ THEN $pval[p] \leftarrow none$
      ELSE $pval[p] \leftarrow v(p)$

$Phase2a(p, r, V) \triangleq$
  pre-conditions:
    $p \in CF(r)$
    $prnd[p] = r$
    $pval[p] = none$
    either ($V \neq Nil \wedge$ $\langle$"propose", $V\rangle_p \Leftarrow p \in Pr$)
    or ($V = Nil \wedge$ $\langle$"2a", $r, \langle q, W\rangle\rangle_{\sigma_q} \Leftarrow q \in CF(r) \wedge W \neq Nil$)
  actions:
    $pval[p] \leftarrow V$
    IF $V \neq Nil$ THEN
      $\langle$"2a", $r, \langle p, V\rangle, proof[p]\rangle_p \Rightarrow A \cup CF(r)$
    ELSE
    $\langle$"2a", $r, \langle p, V\rangle, proof[p]\rangle_p \Rightarrow A$

$Phase2b(a, r) \triangleq$
  LET $Cond1 =$
    $vval[a] = none \vee$
    ($\langle$"2S", $r, v, proofs\rangle_c \Leftarrow C(r) \wedge$
    goodRoundValue(r, v, proofs)
    $v \neq \perp \wedge vrnd[a] < r$)
  LET $Cond2 =$
    $\langle$"2a", $r, \langle p, V\rangle, proofs\rangle_p \Leftarrow p \in CF(r) \wedge V \neq Nil$
    goodRoundValue(r, V, proofs)
  pre-conditions:
    $a \in A$
    $rnd[a] \leq r$
    $Cond1 \vee Cond2$
  actions:
    IF $Cond1$
      THEN $vval[a] \leftarrow v$
      ELSE
        IF $Cond2 \wedge (vrnd[a] < r \vee vval[a] = none)$
          THEN $vval[a] \leftarrow \perp \bullet \langle p, V\rangle \bullet [\langle p, Nil\rangle : p \in Pr \setminus CF(r)]$
          ELSE $vval[a] \leftarrow vval[a] \bullet \langle p, V\rangle$
    $rnd[a] \leftarrow vrnd[a] \leftarrow r$
    $\langle$"2b", $r, vval[a]\rangle_a \Rightarrow L$

$Learn(l) \triangleq$
  pre-conditions:
    $l \in learners$
    $\exists Q \subseteq A$:
      Q is a quorum
      $\forall a \in Q$, $\langle$"2b", $r, \_\rangle_a \Leftarrow a$
  actions:
    $Q2bVals = \{v : \langle$"2b", $r, v\rangle_a \Leftarrow a \in Q\}$
    $w = \sqcap Q2bVals$
    $learned[l] = learned[l] \sqcup w$

cada agente. Isso garante que mensagens diferentes não podem ter o mesmo identificador, que uma próxima mensagem enviada não pode ter um identificador menor que o anterior, e que o próximo identificador gerado é o sucessor direto do último gerado. Essas propriedades garantem que mesmo que o agente esteja comprometido (falho), ele deve executar o serviço da mesma maneira.

A interface do serviço possui duas funções:

***createUI(m)*:** retorna um certificado USIG que contém um identificador único UI e a prova de que este identificador foi criado por este componente para a mensagem *m*. O identificador é basicamente um contador monotonicamente crescente o qual é incrementado a cada vez que a função é chamada. A certificação envolve encriptação e pode ser baseada em um criptografia hash ou criptografia de chaves públicas.

***verifyUI(PK, UI, m)*:** verifica se o identificador único UI é válido para a mensagem *m*.

2.11 USIG-BCFABCAST - USIG-Bizantine Collision-fast Atomic Broadcast

O USIG-BCFABCAST é o algoritmo proposto em [1] que consegue ser tolerante a falhas Bizantinas e Collision-Fast. Ele consegue fazer isso por ser baseado no BCFABCAST e utilizar o componente seguro USIG. Por utilizá-los, este protocolo precisa de apenas 2f+1 *acceptors* e 2 passos de comunicação, sendo rápido a despeito de colisões mesmo na presença de processos maliciosos.

O ponto principal é que atráves dos identificadores únicos adicionados as mensagens, os *CF-proposers* não precisam enviar suas mensagens através dos *acceptors* quando não possuirem nada para propor, isto é, os *learners* podem confiar nas mensagens entregues pelos *proposers* que possuem valores nulos, pois eles não conseguem enviar propostas diferentes com o mesmo identificador.

As fases iniciais de configuração de um novo *round* pelo coordenador não precisam de grandes alterações, pois caso o coordenador envie diferentes valores para o *round r* nas mensagens *1a*, ele não vai conseguir uma prova para reconfigurar o sistemas com as mensagens *2S*. Isso acontece porque as mensagens *2S* contém um identificador único gerado pelo USIG do coordenador e impede que ele possa enviar mensagens *2S* diferentes para agentes diferentes. Portanto, todos os agentes recebem a mesma configuração para o *round r*.

Cada *CF-proposer* também adiciona um identificador único às suas propostas (ação *2a*), o que impede que propostas diferentes sejam enviadas para agentes diferentes. Ape-

sar de essas mensagens serem enviadas para todos os agentes($A \cup CF(r) \cup L$), apenas os *learners* processam as propostas com *Nil*. Isso é necessário para que os agentes incrementem seus contadores para as propostas esperadas, mesmo quando a proposta é *Nil*. Por outro lado, os *learners* incrementam seus contadores para propostas diferentes de *Nil* quando as recebem dos *acceptors*. É importante notar que os identificadores únicos sempre acompanham suas respectivas propostas, de modo que sempre que uma nova rodada iniciar, eles são enviados dos *acceptors* para o coordenador que as irá enviar nas mensagens *2S*. É Importante destacar que os *acceptors* também adicionam um identificador único as mensagens *2b* enviadas aos *learners*, de forma que um *acceptor* não é capaz de enviar mensagens diferentes para diferentes *learners* sem ser descoberto.

Com o uso dos identificadores sequenciais únicos, o protocolo proposto impede que agentes maliciosos enviem mensagens diferentes para diferentes agentes em um mesmo passo do algoritmo. Com isso, ele possui um funcionamento semelhante ao [2], o qual suporta apenas falhas por *crash*. Portanto, cada *CF-proposer* pode fazer apenas uma única proposta para uma determinada rodada r. Caso a proposta seja *Nil*, ela pode ser enviada diretamente para os *learners*, e caso contrário, deve ser enviada para os *acceptors* para que fique armazenada caso seja necessário um novo *round* para finalizar a instância do consenso.

Dessa forma, o algoritmo precisa de apenas 2 passos de comunicação para que um valor proposto possa ser decidido, sendo rápido à despeito de colisões mesmo com a presença de agentes bizantinos. Além disso, somente *2f+1 acceptors* são necessários, pois os mesmos não conseguem modificar uma proposta feita por um *CF-proposer*, devido ao idenfiticador único gerado que acompanha a proposta, e também não conseguem enviar mensagens diferentes para diferentes learners. O Algoritmo 4 mostra a implementação do algoritmo em TLA+.

Para facilitar a comparação dos algoritmos apresentados, a Tabela 2.1 mostra algumas métricas entre eles. Sendo que as 3 últimas colunas são referentes aos protocolos implementados nesse trabalho.

	Paxos	Fast Paxos	CFABCast	BCFABCast	USIG-BCFABCast
Passos	3	2	2	3	2
#Acceptors	2f+1	3f+1	2f+1	5f+1	2f+1
#Quorum	f+1	2f+1	f+1	4f+1	f+1
Falha	Crash	Crash	Crash	Bizantina	Bizantina
Componente	-	-	-	-	USIG

Tabela 2.1: Comparação entre algoritmos

Algorithm 4 USIG *Byzantine Collision-fast Atomic Broadcast* [Saramago et al 2017] [1]

All variables and sets from Algorithm 3
$USIG^C[c]$, $cnt[c]$: USIG used by coordinator c and expected counter value for msgs received from c, initially 0.
$USIG^A[a]$, $cnt[a]$: USIG used by acceptor a and expected counter value for msgs received from a, initially 0.
$USIG^P[p]$, $cnt[p]$: USIG used by CF proposer p and expected counter value for msgs received from p, initially 0.

1: $Propose(p, V) \triangleq$ lines 2-5 of Algorithm 3

2: $Phase1a(c, r) \triangleq$ lines 7-13 of Algorithm 3

3: $Phase1b(a, r) \triangleq$ lines 15-21 of Algorithm 3

4: $Phase2Start(c, r) \triangleq$
5: **pre-conditions:**
6: $c = C(r)$
7: $crnd[c] = r$
8: $cval[c] = none$
9: $\exists Q \subseteq A$:
10: Q is a quorum
11: $\forall a \in Q$, $\langle$"1b", $r, vrnd, vval\rangle_a \Leftarrow a$
12: **actions:**
13: LET $msgs = [\ m = \langle$"1b", $r, vrnd, vval\rangle_a : m \Leftarrow a \in Q\ ]$
14: LET $k = Max([vrnd : \langle$"1b", $r, vrnd, vval\rangle_a \in msgs\ ])$
15: LET $S = [vval : \langle$"1b", $r, k, vval\rangle_a \in msgs, vval \neq none]$
16: IF $S = \emptyset$ THEN
17: $cval[c] \leftarrow \perp$
18: ELSE
19: $cval[c] \leftarrow \sqcup S \bullet [\langle p, Nil, Nil\rangle : p \in CF(r)]$
20: $UI_c \leftarrow USIG^C[c]$.createUI($\langle$"2S", $r, cval[c], msgs\rangle$)
21: $\langle$"2S", $r, cval[c], msgs, UI_c\rangle \Rightarrow CF(r) \cup A$

22: $Phase2Prepare(p, r) \triangleq$
23: **pre-conditions:**
24: $p \in CF(r)$
25: $prnd[p] < r$
26: $\langle$"2S", $r, v, proof, UI_{C(r)}\rangle \Leftarrow C(r)$
27: goodRoundValue(r,v,proofs)
28: verifyUI($PK_{C(r)}, UI_{C(r)}, \langle$"2S", $r, v, proofs\rangle$)
29: verifyCnt($UI_{C(r)}, cnt[C(r)]$)
30: **actions:**
31: $prnd[p] \leftarrow r$
32: IF $v = \perp$ THEN $pval[p] \leftarrow none$
33: ELSE $pval[p] \leftarrow v(p)$

34: $Phase2a(p, r, V) \triangleq$
35: **pre-conditions:**
36: $p \in CF(r)$
37: $prnd[p] = r$
38: $pval[p] = none$
39: **either** $(V \neq Nil \wedge \langle$"propose", $V\rangle_{\sigma_p} \Leftarrow p \in Pr)$
40: **or** $(V = Nil \wedge \langle$"2a", $r, \langle q, W\rangle, UI_q\rangle \Leftarrow q \in CF(r)$
41: verifyUI($PK_q, UI_q, \langle q, W\rangle$) $\wedge$ verifyCnt($UI_q, cnt[q]$) $\wedge W \neq Nil$)
42: **actions:**
43: $pval[p] \leftarrow V$
44: $UI_p \leftarrow USIG^P[p]$.createUI($\langle p, V\rangle$)
45: $\langle$"2a", $r, \langle p, V\rangle, UI_p\rangle \Rightarrow A \cup CF(r) \cup L$

46: $Phase2b(a, r) \triangleq$
47: LET $Cond1 =$
48: $vval[a] = none \vee (\langle$"2S", $r, v, proofs, UI_{C(r)}\rangle \Leftarrow C(r) \wedge$
49: verifyUI($PK_{C(r)}, UI_{C(r)}, \langle$"2S", $r, v, proofs\rangle$) $\wedge$
50: verifyCnt($UI_{C(r)}, cnt[C(r)]$) $\wedge$ goodRoundValue(r,v,proofs) $\wedge$
51: $v \neq \perp \wedge vrnd[a] < r$)
52: LET $Cond2 =$
53: $\langle$"2a", $r, \langle p, V\rangle, UI_p\rangle \Leftarrow p \in CF(r) \wedge V \neq Nil \wedge$
54: verifyUI($PK_p, UI_p, \langle p, V\rangle$) $\wedge$ verifyCnt($UI_p, cnt[p]$)
55: **pre-conditions:**
56: $a \in A$
57: $rnd[a] \leq r$
58: $Cond1 \vee Cond2$
59: **actions:**
60: IF $Cond1$
61: THEN $vval[a] \leftarrow v$
62: ELSE
63: IF $Cond2 \wedge (vrnd[a] < r \vee vval[a] = none)$
64: THEN $vval[a] \leftarrow \perp \bullet \langle p, V, UI_p\rangle \bullet [\langle p, Nil, Nil\rangle : p \in Pr \setminus CF(r)]$
65: ELSE $vval[a] \leftarrow vval[a] \bullet \langle p, V, UI_p\rangle$
66: $rnd[a] \leftarrow vrnd[a] \leftarrow r$
67: $UI_a \leftarrow USIG^A[a]$.createUI($\langle$"2b", $r, vval[a]\rangle$)
68: $\langle$"2b", $r, vval[a], UI_a\rangle \Rightarrow L$

69: $Learn(l) \triangleq$
70: **pre-conditions:**
71: $l \in learners$
72: $\exists Q \subseteq A$:
73: Q is a quorum
74: $\forall a \in Q$, $\langle$"2b", $r, \_, UI_a\rangle \Leftarrow a \wedge$
75: verifyUI($PK_a, UI_a, \langle$"2b", $r, \_\rangle$) $\wedge$
76: verifyCnt($UI_a, cnt[a]$)
77: **actions:**
78: LET $P \subset CF(r) : \forall p \in P$, $\langle$"2a", $r, \langle p, Nil\rangle, UI_p\rangle \Leftarrow p \wedge$ verifyUI($PK_p, UI_p, \langle p, Nil\rangle$) $\wedge$ verifyCnt($UI_p, cnt[p]$)
79: $Q2bVals = [v : \langle$"2b", $r, v, UI_a\rangle \Leftarrow a \in Q \wedge$
80: $\forall \langle q, W, UI_q\rangle \in v$: verifyUI($PK_q, UI_q, \langle p, W\rangle$) $\wedge$
81: verifyCnt($UI_q, cnt[q]$]
82: $w = \sqcap Q2bVals \bullet [\langle u, Nil, UI_u\rangle : u \in P]$
83: $learned[l] = learned[l] \sqcup w$
84: $verifyCnt(UI_x, cnt[x]) \triangleq$
85: IF $(UI_x.cnt = cnt[x])$ THEN
86: $cnt[x] \Leftarrow cnt[x] + 1$
87: return TRUE;
88: ELSE
89: return FALSE

Capítulo 3

Implementação

Nesse capítulo será explicado como foi feita a implementação dos protocolos, qual a linguagem utilizada, e como foi definida a arquitetura da solução proposta. E também serão discutidos quais motivos levoram a cada decisão.

3.1 Linguagem utilizada

Foi decidido utilizar a linguagem Java para implementar os protocolos. A linguagem Java possui uma grande quantidade de APIs públicas, tendo comunidade forte e ativa, existem muitas pessoas que utilizam essa linguagem e isso permite que o protocolo implementado nela possa ser facilmente utilizado por outras pessoas. Outra vantagem é que a linguagem utiliza a JVM, o que permite que um código desenvolvido em determinado SO possa ser compilado e utilizado em outro.

3.2 Maven

O Maven é uma ferramenta de automação e compilação de projetos Java. Seu principal uso foi para o gerenciametno de pacotes, e compilação do programa [22].

3.3 Netty

Netty é um framework para comunicação assíncrona de processos e aplicações. Ele fornece uma abstração de alto nível que permite um desenvolvimento rápido de aplicações que se comunicam em rede [23].

3.4 Dynamic Quoruns

Dynamic Quoruns é uma biblioteca que foi criada utilizando o Netty, ela foca em fornecer métodos ainda mais simples e diretos para comunicação entre os agentes. Para isso, ela se foca em duas coisas: enviar e receber mensagens.

Para se focar nisso, ela possui duas classes principais: QuorumSender e QuorumReplica. A comunicação entre agentes é sempre feita utilizando a classe QuorumMessage que encapsula os dados que são enviados. Além disso, ela utiliza dois arquivos para definir os quoruns, o hosts.config e system.config, que é onde estão configuradas as informações necessárias para os agentes identificarem uns aos outros.

3.4.1 QuorumMessage

Essa classe representa a mensagem que é trocada entre os agentes, ela serve como um container do que será enviado, possuindo dois parâmetros, o identificador do tipo de mensagem (sempre será do tipo QuorumRequest), e o conteúdo em si que se deseja enviar.

3.4.2 QuorumSender

Essa classe abstrata deve ser extendida por qualquer classe que queira enviar mensagens para os outros agentes, o envio de mensagens é feito a partir do metodo sendTo, onde deve ser especificado para quais agentes a mensagem será enviada.

3.4.3 QuorumReplica

Essa classe abstrata deve ser estendida por qualquer classe que deseja receber uma mensagem, ou seja, quando uma mensagem é enviada por um *QuorumSender*, deve existir algum *QuorumReplica* para receber essa mensagem. Essa classe possui dois métodos: *executeRequest*, e *executeReconfigurationMessage*. Nos protocolos executados somente é implementado o método *executeRequest*, pois não utilizaremos o método *executeReconfigurationMessage*, que está relacionado com a reconfiguração do conjunto de replicas (os protocolos estudados não suportam reconfiguração).

3.4.4 Arquivos necessários

Para a identificação dos endereços dos agentes (id, IP, porta) é necessario criar o arquivo hosts.config, ele é responsavel pela identificação das classes que instânciam o *QuorumReplica*, e o arquivo system.config, ele é responsável pela identificação das classes que instânciam o *QuorumSender*.

Com isso, cada agente possui a própria implementação do QuorumSender e do QuorumReplica para que eles possam se comunicar.

3.5 Arquitetura

Cada um dos protocolos (CFABCAST, BCFABCAST, e USIG-BCFABCAST) funcionam de maneira similar, onde existe um cliente e os agentes do protocolo, o cliente envia uma mensagem para um *proposer* e espera a confirmação.

Foi criado uma classe para cada agente do protocolo, e cada classe possui sua própria implementação de *QuorumSender* e de *QuorumReplica* para que todos possam enviar e receber mensagens. Além dos agentes do protocolo, também temos o cliente, que possui as mesmas implementações, mas suas únicas ações são de enviar mensagens para serem propostas, e receber a confirmação de que elas foram aprendidas.

Todos os agentes do protocolo e o cliente herdam da classe Agente, é nessa classe que estão configuradas as informações sobre os outros agentes, ou seja, por herdar dessa classe, todos os agentes conseguem encontrar a referência aos outros.

Cada agente no protocolo tem uma função especifica, na implementação cada agente possui um método que corresponde a essa função. Por exemplo, o proposer executa a ação *phase1A* no protocolo, então a classe Proposer possui um mètodo correspondente com o mesmo nome, e isso ocorre para cada agente do protocolo.

3.6 CFABCAST

O protocolo foi feito utilizando o Fast-Paxos como base, então ele possui suas configurações como base, como já dito anteriormente, cada agente tem sua própria implementação dos passos que ele executa no protocolo. O protocolo é iniciado quando o cliente envia uma mensagem para o proposer, com isso ele cria uma nova instância do protocolo e da início ao processo de decisão. A mensagem enviada pelo cliente é uma *ClientMessage* que ao ser recebida pelo *Proposer* é encapsulada por uma *ProtocolMessage* que é utilizada durante toda a execução do protocolo.

Para implementar o *v-map* foi utilizada a classe java Map que já possui a abstração necessária dessa estrutura, sendo feito então o mapa do id do *Proposer* para a mensagem *ClientMessage* recebida. Esse mapa é criado pelos *acceptors* ao receber uma proposta vindo de um *proposer* e é então enviado para os *learners*, eles então fazem a comparação desse mapa, verificando se todos, ou a maioria, apontam para o mesmo valor, isso é feito fazendo a comparação entre os valores de cada *v-map* recebido.

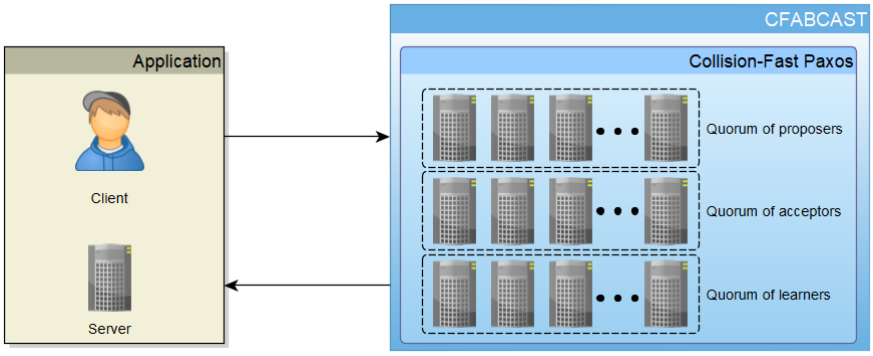

Figura 3.1: Modelo da arquitetura da implementação do CFABCAST

Com o *learner* decidindo o valor a ser aprendido, ele o coloca em uma lista com os valores aprendidos, para que possa ser referenciado futuramente, se necessário.

A Figura 3.1 mostra como funciona a arquitetura do CFABCAST, sendo o protocolo feito com base no ABCAST, e se comunicando com a aplicação cliente, que envia e recebe mensagens para os agentes do protocolo.

O algoritmo implementado pode ser encontrado no endereço: `https://github.com/VMateus00/TCC`.

3.7 BCFABCAST

O BCFABCAST foi implementado tendo como base o CFABCAST, ou seja, as classes dele estendem das classes do CFABCAST, isso pode ser visto a partir da Figura 3.2. Nele, as classes agentes são BProposer, BAcceptor e BLearner, pois como dito, eles estendem o modelo do CFBACAST, com métodos a mais que permitem que eles sejam tolerantes a falhas bizantinas. Além disso, eles implementam a interface BAgent, que é responsável por assinar as mensagens enviadas e verificar se a assinatura da mensagem é válida.

A interface BAgent possui dois métodos principais: *createAssignedMessage*, que é responsável por criar e assinar uma mensagem do agente, utilizando os parâmetros necessários do protocolo e a chave privada dele, e o método *verifyMsg* que recebe uma mensagem e faz a verificação se a assinatura dela é válida, todos os agentes executam esse método quando recebem uma mensagem para poder validá-la.

Nesse protocolo, também foi criada uma mensagem própria, chamada *BProtocolMessage*, que extende de *ProtocolMessage*, sua diferença é que ela carrega, além dos parâmetros da *ProtocolMessage* a assinatura e as provas que são entregues para provar que o agente executou o passo corretamente.

A implementação do algoritmo pode ser encontrada no endereço: `https://github.com/VMateus00/TCC`.

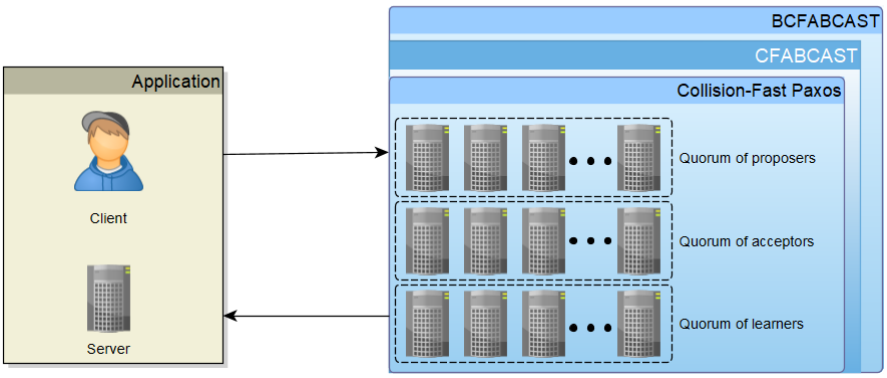

Figura 3.2: Modelo da arquitetura da implementação do BCFABCAST

3.8 USIG-BCFABCAST

Como mostrado em [1], o USIG-BCFABCAST é implementado utilizando como base o BCFABCAST, onde ele utiliza o componente USIG, já explicado anteriormente, e a assinatura digital, sua arquitetura pode ser vista na Figura 3.3.

Temos então as classes *USIGBProposer*, *USIGBAcceptor* e *USIGBLearner* que herdam as classes *BProposer*, *BAcceptor* e *BLearner*, respectivamente, vindas do BCFABCAST. Cada classe possui sua própria implementação do componente USIG, que garante que cada um tem seu respectivo contador, ou seja, o componente seguro foi emulado em software, e não utilizado um software já existente para isso. O protocolo possui também sua própria mensagem de comunicação, ela é a *USIGBProtocolMessage* e ela é uma classe filha de *BProtocolMessage*, tendo além dos atributos da super classe, os atributos para carregar o identificador USIG.

Cada classe possui a própria implementação da classe *UsigComponent*, essa classe possui dois métodos, são eles: *createUI* que é responsável por criar um *USIGBProtocol-Message* assinado, e o método *verifyUI* que verifica se a assinatura do componente USIG é válida.

É importante ressaltar que todos os agentes que vão em algum momento receber uma mensagem de um agente x, devem sempre receber todas as mensagens que forem enviadas pelo agente x, para que possam atualizar o contador USIG referente a esse agente, pois se em algum momento ele enviar uma mensagem e o agente não tiver recebido a mensagem anterior, ele não poderá dar prosseguimento ao progresso do protocolo.

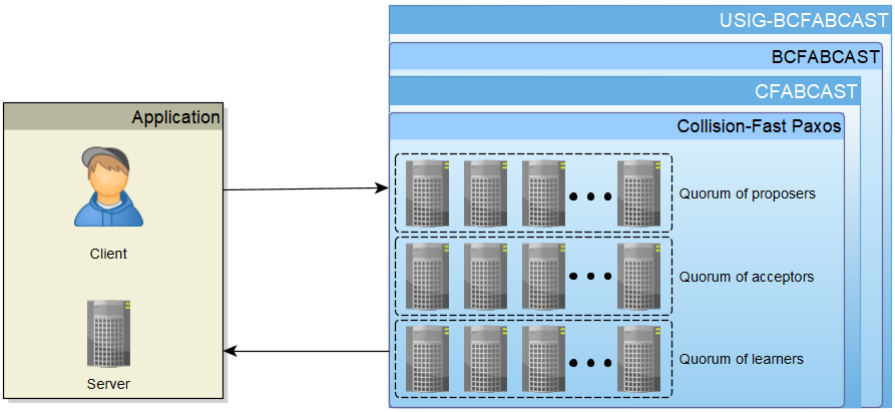

Figura 3.3: Modelo da arquitetura da implementação do USIG-BCFABCAST

Capítulo 4

Experimentos

Os experimentos foram feitos utilizando o ambiente Emulab [24], esse ambiente é público e disponibiliza uma infra-estrutura de teste para pesquisadores do mundo todo, além de ferramentas que auxiliam o desenvolvimento, avaliação e depuração de sistemas. O Emulab permite a criação de uma topologia de rede configurável e um ambiente controlado, podendo definir a largura de banda e latência, e que permitindo acesso total a nó computacional.

O ambiente utilizado consiste de 16 (dezesseis) máquinas conectadas através de uma rede *Gigabit Ethernet* em topologia estrela, com velocidade de comunicação de 100Mbs e sem interferência entre os nós, como pode ser observado na figura 4.1. Cada nó possui um processador Intel Xeon 3.00 GHz utilizando sistema operacional Ubuntu 14.04.

Nos experimentos cada agente ficou exclusivamente em uma maquina, para garantir que cada agente não concorra por processamento com outro na mesma máquina. Nesses experimentos estamos interessados em obter dois dados: a latência, e o *throughput*. A latência é tempo de resposta entre o envio e o recebimento da confirmação de que o comando foi aprendido, ou seja, é o tempo que leva para o cliente receber a confirmação da mensagem que ele enviou. O *throughput* é relacionado ao *learner*, sendo quantas mensagens ele aprendeu em um determinado intervalo de tempo. Nestes experimentos todos os dados serão avaliados em milisegundos.

Esses dados permitem verificar o desempenho do sistema, e são inversamente proporcionais, conseguindo assim que um confirme o outro.

4.1 Cenários

Para cada protocolo, o experimento foi repetido 3 vezes, e foi feita a seguinte disposição de agentes: 2 (dois) *CF-Proposers*, tamanho minímo de quorum de *acceptors* de cada protocolo para tolerar uma falha e 1 (um) *learner*. Foi variado a quantidade de clientes,

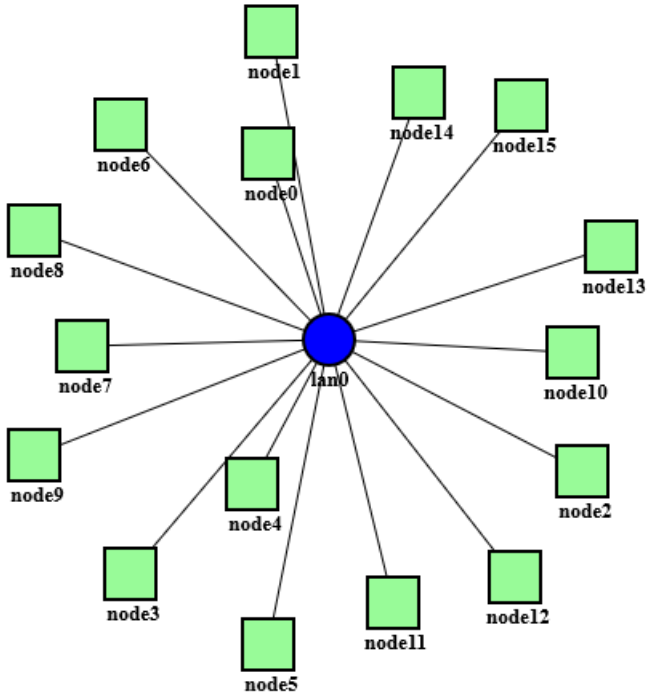

Figura 4.1: Topologia utilizada

sendo realizado de maneira exponencial, sendo 1 cliente, 3 clientes, e por último 5 clientes simultaneamente. Cada cliente enviou inicialmente 10 mensagens para serem aprendidas e após confirmação enviaram mais mensagens, até completar um total de 50 mensagens por cliente.

Os arquivos gerados com os resultados obtidos podem ser acessados no endereço: `https://drive.google.com/drive/u/1/folders/1VyftEjhJ2J0vK4RS8QM-A6Oj3Sv1FUur`. E com os dados obtidos foram geradas as Figuras 4.2 e 4.3.

4.2 Resultados

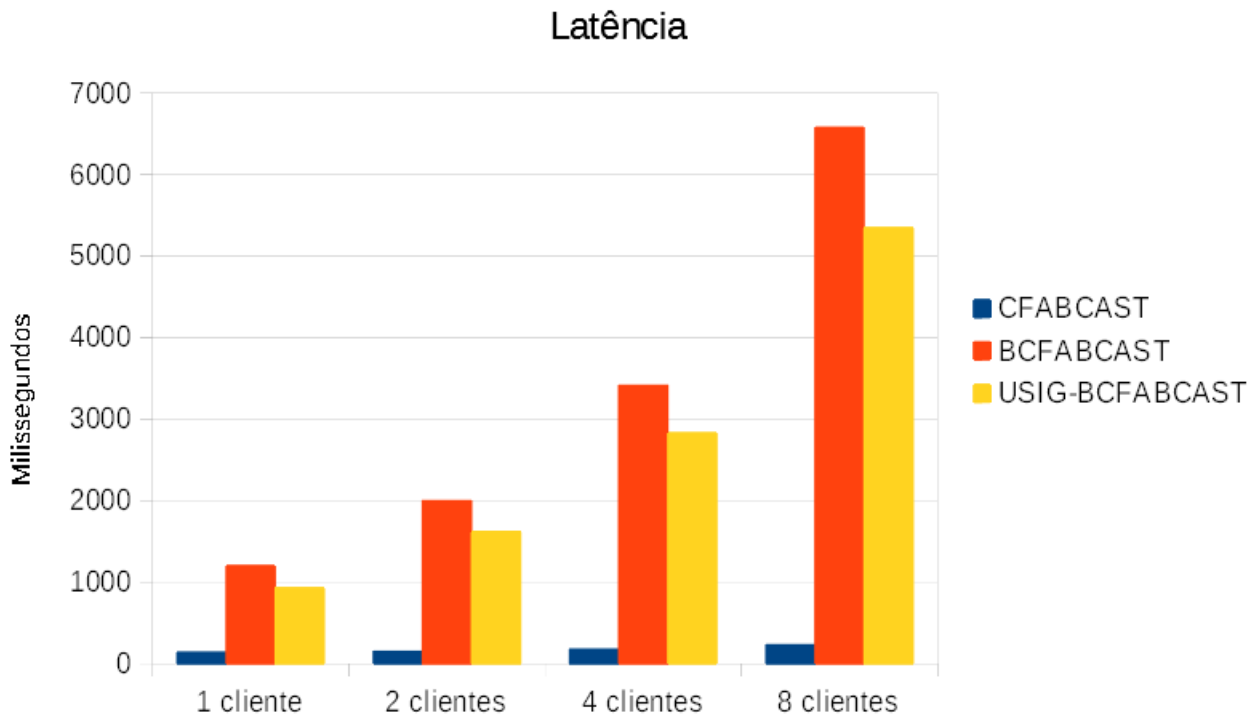

Figura 4.2: Latência obtida em milissegundos

A latência mostra quanto tempo uma mensagem demorou para ser confirmada, isto é, o tempo que um cliente leva para receber a confirmação de sua mensagem. Os dados mostram que o mais rápido, ou seja, que possui a menor latência é o algoritmo CFABCAST, tendo uma eficiência de mais de 50% comparado com os outros algoritmos e apesar da latência parecer igual em todos os casos, isso se deve a escala utilizada, pois a variação dos outros dados é muito grande, o que impede uma melhor visualização, mas em cada caso há um aumento de aproximadamente 10 milissegundos na latência. O BCFABCAST se mostrou o mais lento, tendo a maior latência, isso quer dizer que ele demora mais para enviar a confirmação das mensagens aprendidas para o cliente. E por último temos o USIG-BCFABCAST que possui uma latência superior ao BCFABCAST sendo aproximadamente 20% mais eficiente do que ele, o que corrobora para a confirmação da teoria esperada.

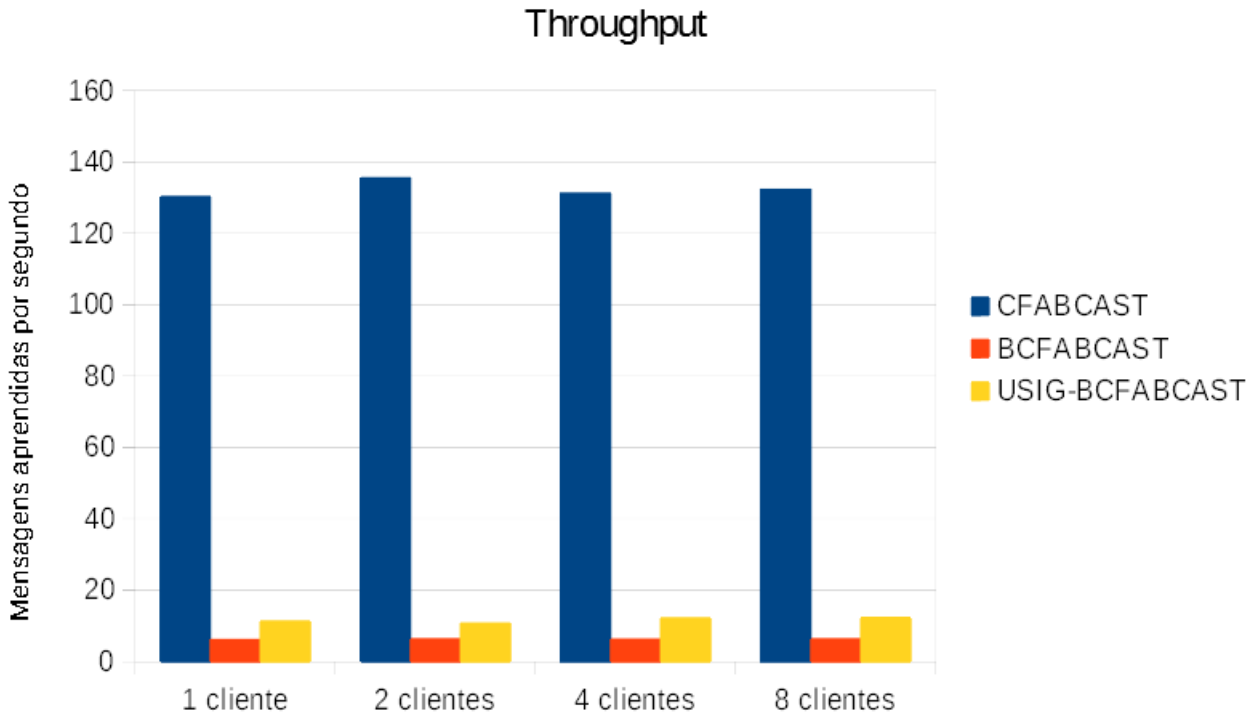

Figura 4.3: Throughput obtido por segundo

O *throughput* é a quantidade de mensagens aprendidas por segundo por um *learner*, sendo assim ele mostra um gráfico inverso em relação ao da latência, nele podemos ver que o CFABCAST é muito superior aos outros dois, aprendendo bem mais mensagens em menos tempo, e o USIG-BCFABCAST é 40% mais eficiente que o BCFABCAST, sendo assim preferivel a ele quanto a utilizar um protocolo que seja tolerante a falhas bizantinas.

É importante notar que em todas as situações apresentadas o CFABCAST apresentou um desempenho superior ao BCABCAST e ao USIG-BCFABCAST, porém ele não é tolerante a falhas bizantinas, sendo essa sua principal fraqueza com relação aos outros, ou seja, apesar do ganho por não ter essas verificações, ele não garante que o protocolo chegue em uma decisão correta caso ocorra falha desse tipo. Com isso temos que o USIG-BCFABCAST se mostrou mais eficiente que o BCFABCAST colaborando com o esperado na teoria.

Capítulo 5

Conclusões e Trabalhos Futuros

5.1 Visão geral

A utilização de sistemas distribuídos está se tornando cada vez mais presente, por isso, tornar os protocolos atuais mais eficientes e robustos é uma necessidade real, a evolução desses protocolos permite que tenhamos sistemas melhores e mais seguros. O USIG-BCFABCAST mostrou-se claramente eficiente e uma boa opção para ser utilizado em sistemas reais, sendo seu uso eficaz em sistemas reais, claramente é necessário que sejam feitos mais experimentos para que possa comparar os resultados aqui obtidos, porém percebe-se o seu grande potencial para uso na indústria.

5.2 Revisão dos objetivos e contribuições

Todos os objetivos propostos foram satisfatoriamente atingidos. Sendo assim, esse trabalho tem como principais contribuições:

- Implementação *open source* dos algoritmos BCFABCAST e USIG-BCFABCAST, permitindo a sua utilização por qualquer pesquisador ou desenvolvedor que o desejar.
- Análise experimental dos protocolos propostos, e discussão sobre os resultados obtidos, o que mostra claramente a eficiêcia dos protocolos propostos e qual a vantagem em utilizá-los.

5.3 Perspectivas futuras

Como perspectivas futuras, temos a replicação dos experimentos utilizando um software que implementa o componente seguro USIG e um próprio para a assinatura de chaves digitais, pois todos esse componentes foram criados manualmente, portanto a utilização

de softwares reais tornaria mais fidedigno o resultado obtido. Logo, podemos definir como passos futuros, a replicação dos experimentos utilizando softwares reais que possam fornecer resultados mais próximos dos reais.

Referências

[1] Saramago, Rodrigo Q., Eduardo A. P. Alchieri, Tuanir F. Rezende e Lasaro Camargos: *Algoritmo de Difusão Atômica Rápido à Despeito de Colisões Tolerante a Falhas Bizantinas*. Belém, Brazil, 2017. 6, 7, 10, 13, 24, 27, 28, 29, 31, 36

[2] Schmidt, Rodrigo, Lasaro Camargos e Fernando Pedone: *Collision-Fast Atomic Broadcast*. maio 2014, ISBN 978-1-4799-3630-4. `https://ieeexplore.ieee.org/document/6838782/`. 10, 13, 18, 20, 22, 23, 24, 30

[3] Hashemi, Mazdak: *Twitter Architeture*, janeiro 2017. `https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html`. 12

[4] Hoff, Todd: *Facebook Architeture*, maio 2017. `http://highscalability.com/blog/2011/5/17/facebook-an-example-canonical-architecture-for-scaling-billi.html`. 12

[5] Prado, Jean: *O que é blockchain?* `https://tecnoblog.net/227293/como-funciona-blockchain-bitcoin/`. 12

[6] Lamport, Leslie: *Time, Clocks, and the Ordering of Events in a Distributed System*. Commun. ACM, 21:558–565, 1978. 12

[7] Lampson, Butler W.: *How to Build a Highly Available System Using Consensus*. Em *Proceedings of the 10th International Workshop on Distributed Algorithms*, WDAG '96, páginas 1–17, Berlin, Heidelberg, 1996. Springer-Verlag, ISBN 3-540-61769-8. `http://dl.acm.org/citation.cfm?id=645953.675640`. 12

[8] Chandra, Tushar Deepak e Sam Toueg: *Unreliable Failure Detectors for Reliable Distributed Systems*. J. ACM, 43(2):225–267, março 1996, ISSN 0004-5411. `http://doi.acm.org/10.1145/226643.226647`. 12, 18

[9] Mao, Yanhua, Flavio Paiva Junqueira e K Marzullo: *Mencius: Building Efficient Replicated State Machine for WANs.* páginas 369–384, 2008. 13

[10] Du, J., D. Sciascia, S. Elnikety, W. Zwaenepoel e F. Pedone: *Clock-RSM: Low-Latency Inter-datacenter State Machine Replication Using Loosely Synchronized Physical Clocks*. Em *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, páginas 343–354, junho 2014. 13

[11] Tanenbaum, Andrew S. e Maarten Van Steen Steen: *Distributed Systems Principles and Paradigms*. Pearson - Prendice Hall, Amsterdam, The Netherlands, segunda edição, 2007. 13, 15, 16

[12] Tanenbaum, Andrew S: *Sistemas Operacionais Modernos*. Pearson Prentice Hall, 2ª edição, 2003. 16

[13] Lamport, Leslie: *Generalized Consensus and Paxos.* 2005. 17

[14] Lamport, Leslie: *The Part-time Parliament*. ACM Trans. Comput. Syst., 16(2):133–169, maio 1998, ISSN 0734-2071. http://doi.acm.org/10.1145/279227.279229. 19

[15] Burrows, Mike: *The Chubby Lock Service for Loosely-coupled Distributed Systems*. Em *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, páginas 335–350, Berkeley, CA, USA, 2006. USENIX Association, ISBN 1-931971-47-1. http://dl.acm.org/citation.cfm?id=1298455.1298487. 19

[16] Lamport, Leslie: *TLA+*, 2018. https://lamport.azurewebsites.net/tla/tla.html. 19

[17] Lamport, Leslie: *Fast Paxos*. Distributed Computing, 19(2):79–103, outubro 2006, ISSN 1432-0452. https://doi.org/10.1007/s00446-006-0005-x. 21

[18] Lamport, Leslie: *Paxos Made Simple*. SIGACT News, 32(4):51–58, 2001, ISSN 0163-5700. http://research.microsoft.com/users/lamport/pubs/paxos-simple.pdf. 21

[19] Correia, M., N. F. Neves e P. Verissimo: *How to tolerate half less one Byzantine nodes in practical distributed systems*. Em *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, páginas 174–183, outubro 2004. 27

[20] Chun, Byung Gon, Petros Maniatis, Scott Shenker e John Kubiatowicz: *Attested Append-only Memory: Making Adversaries Stick to Their Word*. Em *Proc. of Twenty-first ACM SIGOPS Symp. on Operating Systems Principles*, SOSP '07, páginas 189–204, New York, NY, USA, 2007. ACM, ISBN 978-1-59593-591-5. 27

[21] Veronese, Giuliana Santos, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung e Paulo Verissimo: *Efficient Byzantine Fault-Tolerance*. 62(1):16–30, 2013, ISSN 1557-9956. https://ieeexplore.ieee.org/document/6081855/. 27

[22] Apache: *Maven*. https://maven.apache.org/. 32

[23] Apache: *Netty*. https://netty.io/. 32

[24] The University of Utah: *Emulab*. https://www.emulab.net. 38